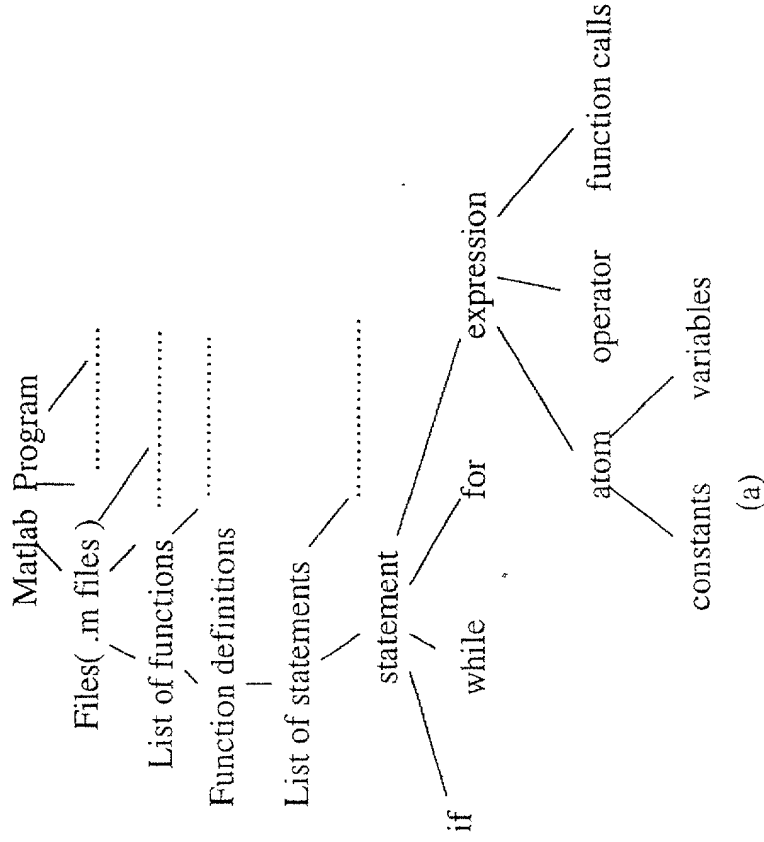


Figure 1: Synthesis flow.



```

pc_filter = rand(63,1);
pc_filter_time = zeros(9512,1);
pc_filter_freq = fft(pc_filter_time);
for i = 1:6
    pc2(i,:) = pc1(i,:)*pc_filter_freq
end;
    
```

(b)

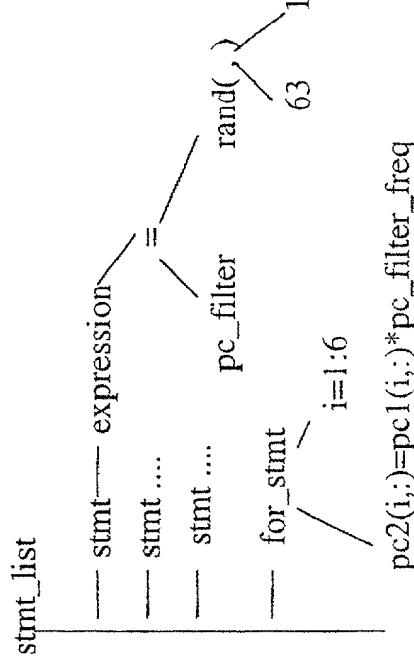


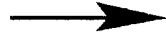
Figure 2: Abstract Syntax Tree: (a) The hierarchy captured by the formal grammar (b) A sample code snippet (c) Abridged syntax tree for the code snippet.

Unlevelized

a = b * c * d;

j = j + a;

i = i + 1;



state 1 : a = b * c * d;

state 2 : j = j + a ;

state 3 : i = i + 1 ;



clock period determined
by the longest state

Execution time = 3 X 30 = 90ns

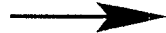
Levelized

t1 = b * c ;

t2 = t1 * d;

j = j + t2 ;

i = i + 1 ;

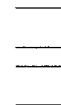


state 1 : t1 <= b * c ;

state 2 : t2 <= t1 * d ;

state 3 : j <= j + t2 ;

state 4 : i <= i + 1 ;



longest state reduced
by levelization

Execution time = 4 X 15 = 60ns

Time to multiply : 15ns

Time to add : 10ns

Execution time = Number of States X Clock Period

Figure 3: Levelization improves clock period.

```

a = 1;
b = 1;
c = a + b;
c = c + 1;

when state 1 => a <= 1;
               next_state <= state 2;

when state 2 => b <= 1;
               next_state <= state 3;

when state 3 => c <= a + b;
               next_state <= state 4;

when state 4 => c <= c + 1;

```

(a)

(b)

Figure 4: (a) Simple MATLAB statements. (b) State machine that steps through the statements sequentially.

```

if( x )
    a = b + 1;
else
    a = b - 2;
end;
c = a + 1;

when state 1 => if( x ) then
    next_state <= state 2;
else
    next_state <= state 3;
when state 2 => a <= b + 1;
    next_state <= state 4;
when state 3 => a <= b - 2;
    next_state <= state 4;
when state 4 =>
    c <= a + 1;

```

(a) (b)

Figure 5: (a) Conditional MATLAB code (b) State machine for Conditional Code.

```

for i = 1 : 256
    a = a + i;
end;

when state 1 => i <= 1;
    next_state <= state 2;

when state 2 => if( i <= 256 ) then
    next_state <= state 3;
else
    next_state <= state 5;
endif;

when state 3 => a <= a + i;
    next_state <= state 4;

when state 4 => i <= i + 1;
    next_state <= state 2;

when state 5 => [ next statement after loop ]

```

(a) (b)

Figure 6: (a) A MATLAB *for* loop (b) State machine for *for* loop.

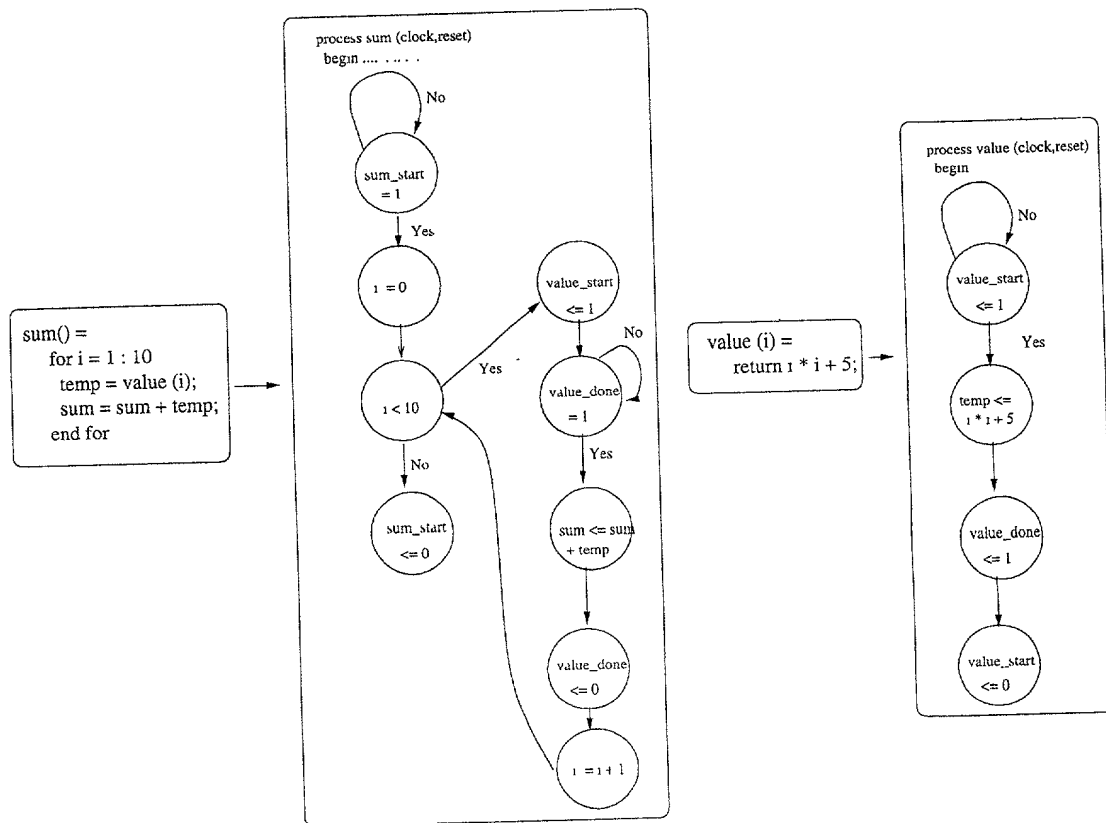


Figure 7: State machine representation of a function call in MATLAB

`a[i + 1] = b + c ;`

↓
Levelization
↓

`t1 = b + c ;`
`t2 = i + 1 ;`
`a[t2] = t1 ;`

```
when state 1 => t1 <= b + c ;
                next_state <= state 2 ;
when state 2 => t2 <= i + 1 ;
                next_state <= state 3 ;
when state 3 => mem_request <= '0' ;
                mem_write_enable <= '0' ;
                next_state <= state 4 ;
when state 4 => mem_address <= Base_a + t2 ;
                mem_data_out <= t1 ;
                next_state <= state 5 ;
when state 5 => if( mem_grant = '0' ) then
                next_state <= state 6 ;
            else
                next_state <= state 4 ;
            endif;
```

(a)

(b)

Figure 8: (a) Array statement in MATLAB and its levelization (b) VHDL corresponding to the MATLAB code. The signals *mem_request*, *mem_data_out*, *mem_grant*, *mem_write_enable* and their particular states and assignments are specified in an external file read by the compiler. *Base_a* is a constant denoting the starting address of the array *a* in memory.

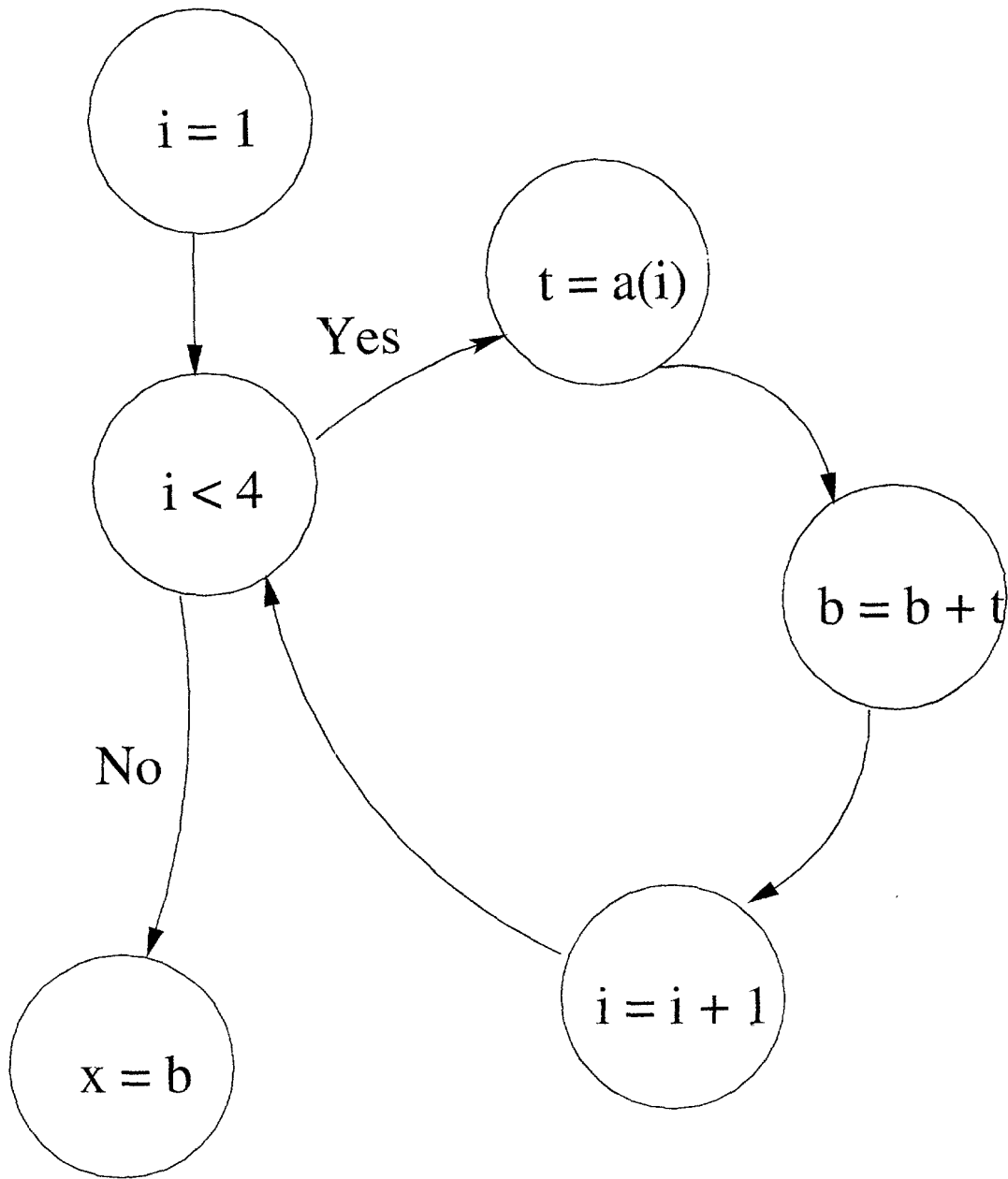


Figure 9: FSM Representation of a code section

Algorithm 1 architecture ...

```

process ...

    if reset = '1' then
        ...
    elsif rising_edge(clock)
        case control is
            when state1 =>
                i := 1;
                control <= state2;
            when state2 =>
                if (i < 4) then
                    ....
                else
                    ....
                endif;
            when state3 =>
                t := a(i);
                control <= state4;
            when state4 =>
                b := b + t;
                control <= state5;
            when state5 =>
                i := i + 1;
                control <= state2;
        end case;
    end if;
end process;

```

Figure 10: VHDL code generated for the above FSM

over manual design or design at a low level of algorithm description (e.g., directly in VHDL).

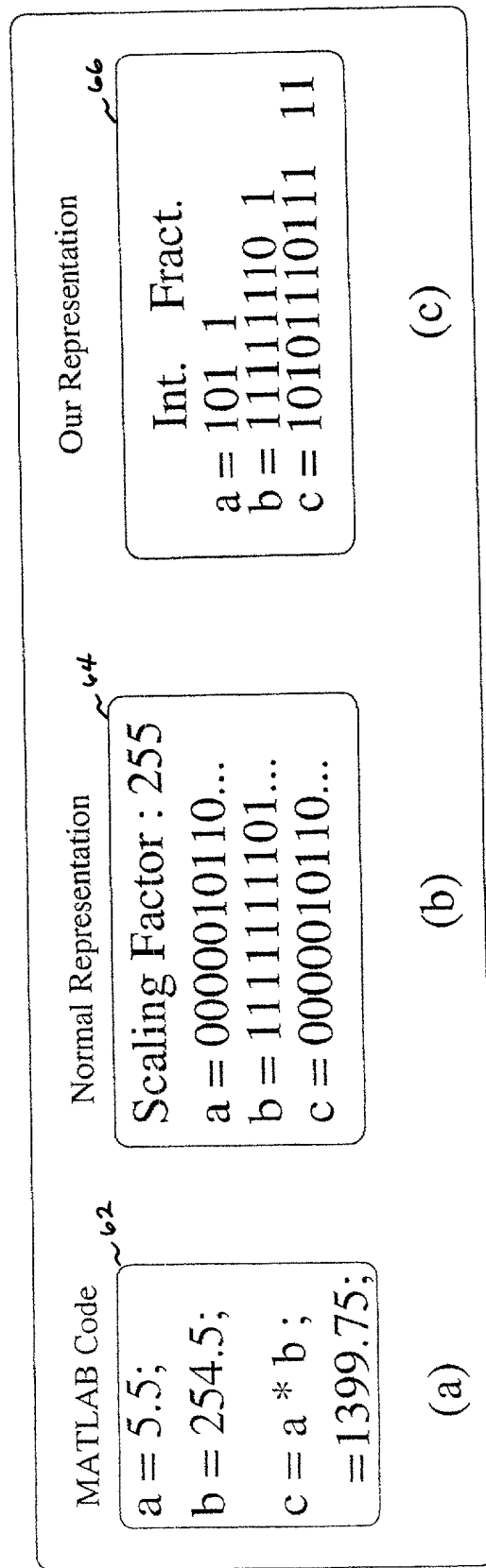


Figure 11: Representation of Real Variables

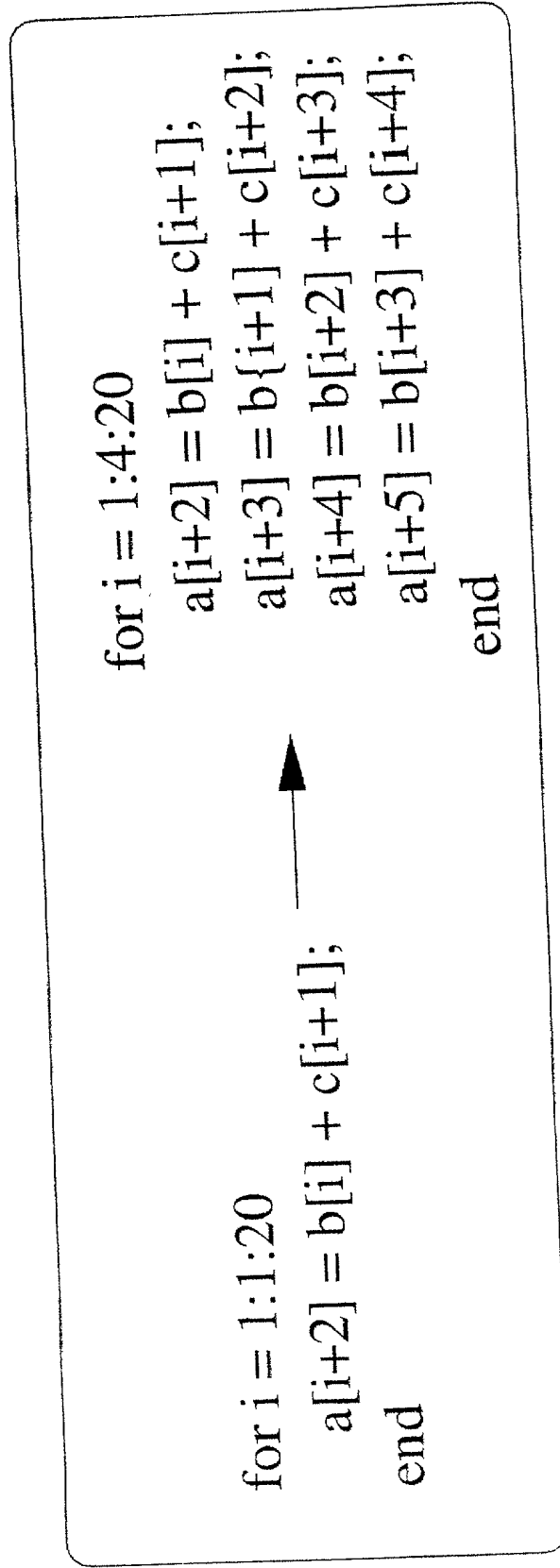


Figure 12: Example showing loop unrolled for Memory Packing

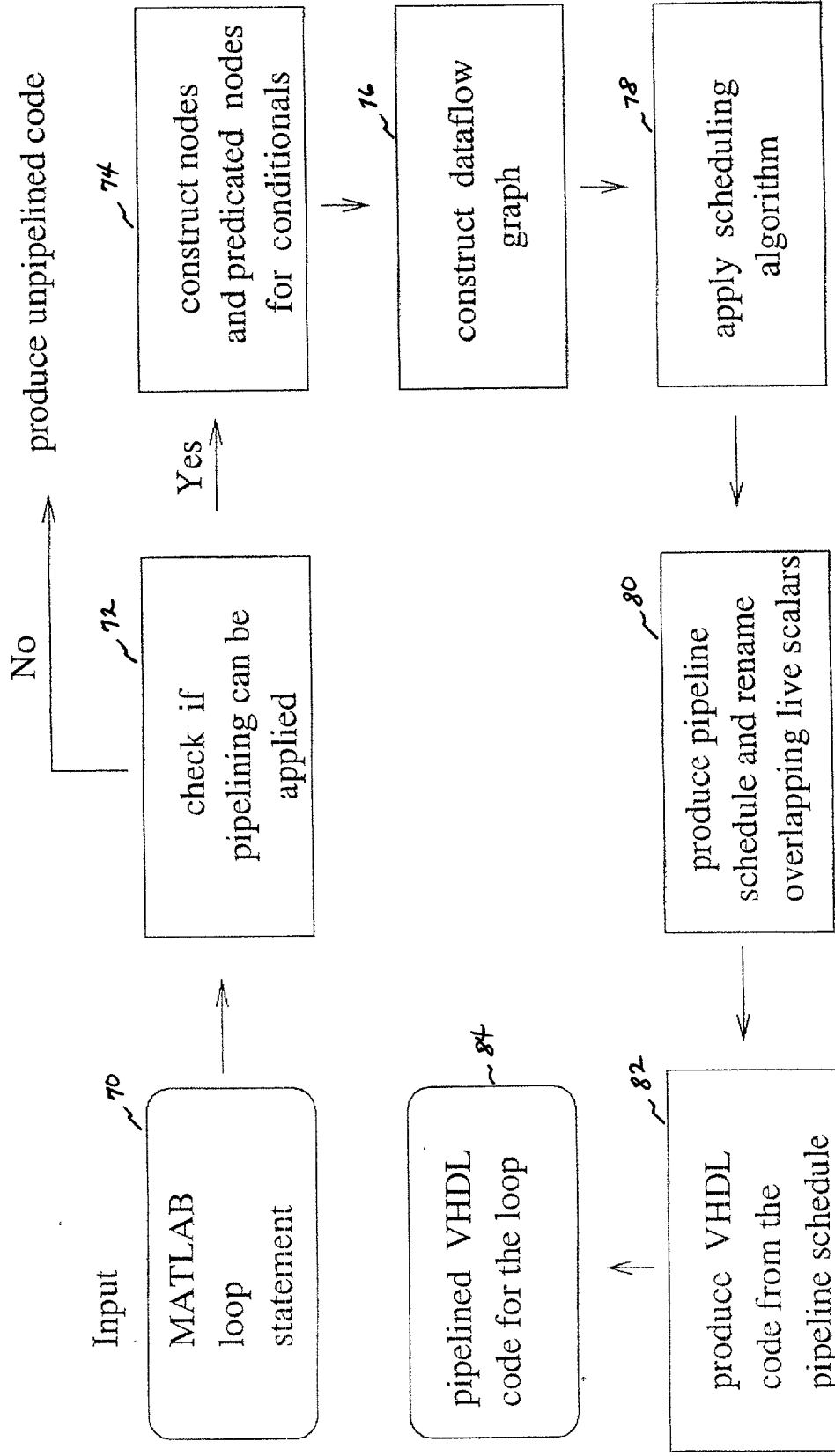


Figure 13: Overall framework for pipelining optimizations

```

for i = x : y : z
    a[i] = a[i] + 2 ;
    x    = a[i - 1] + 3 ;
end ;

```

index variable

index initial value

index step

index limit

Loop Body

Loop Statement

```

if ( i )
    a[i] = a[i] + 1 ;
    x    = a[i + 1] - 4 ;
end ;

```

Conditional Variable

Conditional Statement Body

Conditional Statement

Figure 14: Illustration of Terms used in pipelining framework

Node

$x = x + 1 ;$

Diagram illustrating the concept of a Predicate:

- A **Node** (containing $x = x + 1 ;$) is connected to a **Predicate** (labeled **a**).
- Another **Node** (containing $x = x + 1 ;$) is connected to a **Predicate** (labeled **not a**).

Diagram illustrating the structure of a node and its associated predicates:

- Node**: $x = x + 1 ;$
- Predicates**: a and t

Diagram illustrating the structure of a node and its associated predicates:

- Node**: $x = x + 1 ;$
- Predicates**: $\text{not } a$ and k

Diagram illustrating the structure of a node and its associated predicates:

- Node**: $x = x + 1 ;$
- Predicates**: $\text{not } a$ and $\text{not } k$

Figure 15: Illustration of Construction of nodes from MATLAB statements

Example Array Access Statement

```
x = a(i, j, k);      % a is a 256 X 256 X 256 array
```

```
s1 : ad_temp3 := i * 256 * 256 ;
s2 : ad_temp2 := ad_temp3 + j * 256 ;
s3 : ad_temp1 := ad_temp2 + k ;
s4 : array_address := BaseAddress_a + ad_temp1 ;
```

Address
Calculation

```
s5 : mem_request <= '0' ;
      mem_write_enable <= '0' ;
s6 : mem_address <= array_address ;
s7
s8 : x := mem_data_in ;
```

Memory
Interface
Specific
Signals

Example Memory Access Specification File	
Read	
s1 :	mem_request <= '0' ; mem_write_enable <= '0' ;
s2	mem_address <= [MEMADDRESS]
s3	
s4	[MEMDATA] <= mem_data_in

Figure 16: Example of node construction for array access statements

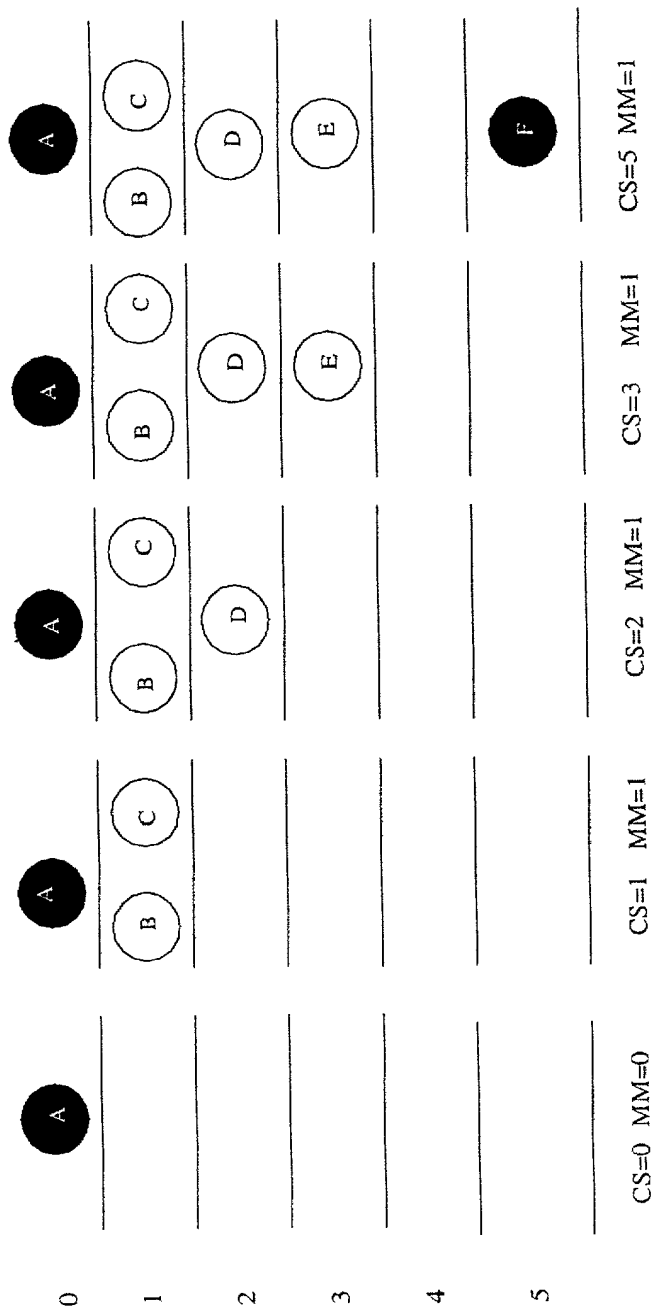


Figure 17: An illustration of pipeline method

105210" 1490460

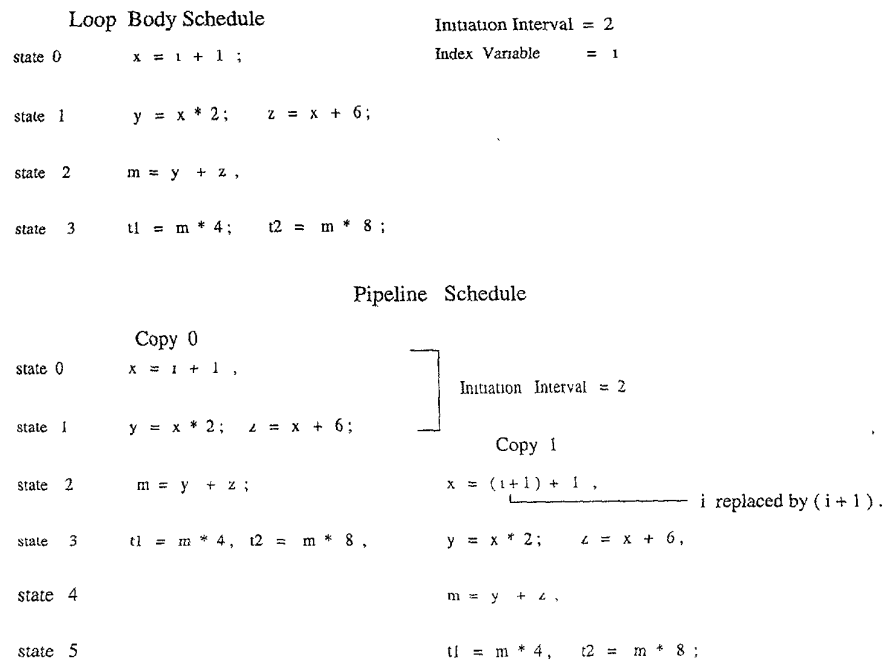


Figure 18: Construction of pipeline schedule from loop body schedule

Loop Body Schedule

```
s1  x = i + 1 ;
s2  y = i * 2 ,
s3  k = x + 3 ,
```

Pipeline Schedule

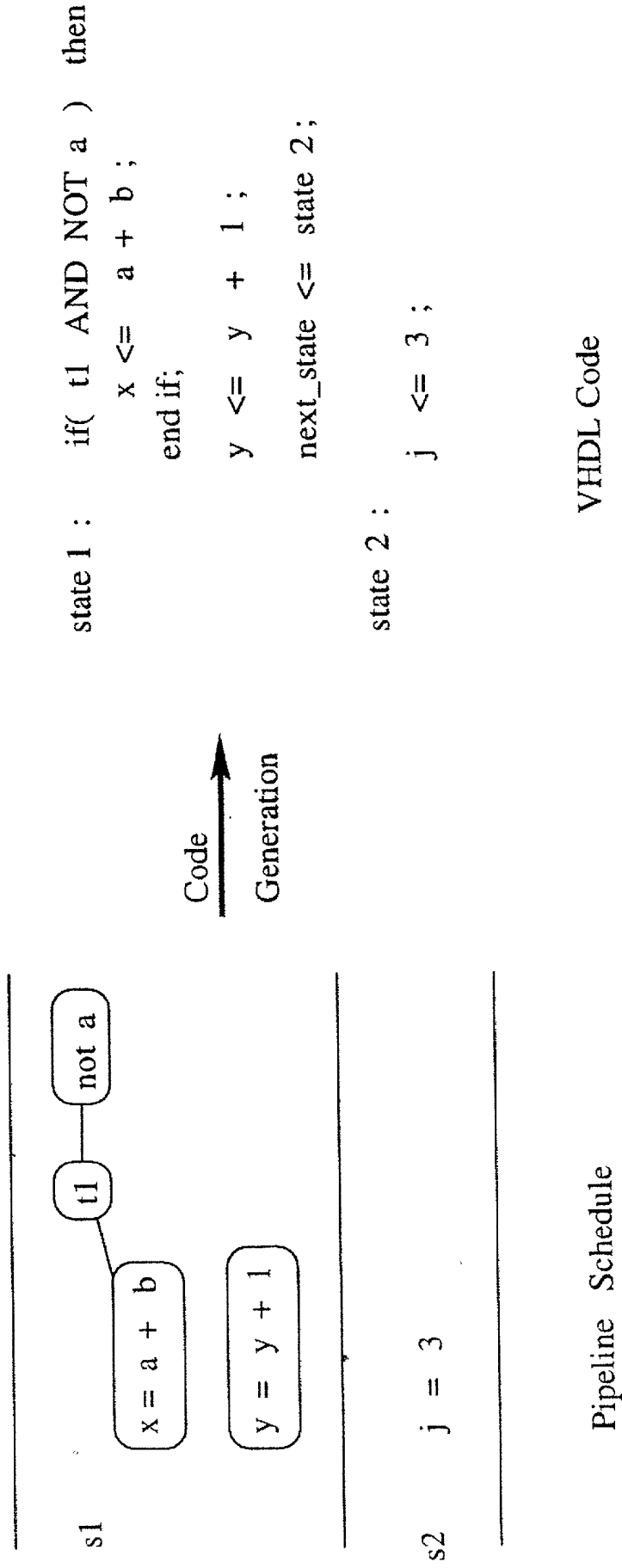
	Copy 0	Copy 1	Copy 2
s1	x = i + 1 ;		
s2	y = i * 2 ;	x = i + 1 ;	
s3	k = x + 3 ;	y = i * 2 ;	x = i + 1 ;
s4		k = x + 3 ;	y = i * 2 ;
s5			k = x + 3 ;

Vertical arrows labeled "x live range" indicate the live range of variable x across the pipeline stages.

Pipeline Schedule with Overlapping Live Scalars Renamed

	Copy 0	Copy 1	Copy 2
s1	case mod_var is when 0 : x0 = i + 1 ; when 1 : x1 = i + 1 ; when 2 : x2 = i + 1 ; end case ;		
s2	y = i * 2 ;	case mod_var is when 0 : x1 = i + 1 ; when 1 : x2 = i + 1 ; when 2 : x0 = i + 1 ; end case ;	
s3	case mod_var is when 0 : k = x0 + 3 ; when 1 : k = x1 + 3 ; when 2 : k = x2 + 3 ; end case ,	y = i * 2 ;	case mod_var is when 0 : x2 = i + 1 ; when 1 : x0 = i + 1 ; when 2 : x1 = i + 1 ; end case ;
s4		case mod_var is when 0 : k = x1 + 3 ; when 1 : k = x2 + 3 ; when 2 : k = x0 + 3 ; end case ,	y = i * 2 ;
s5			case mod_var is when 0 : k = x2 + 3 ; when 1 : k = x0 + 3 ; when 2 : k = x1 + 3 ; end case ,

Figure 19: Renaming of scalars with live overlapping ranges in the pipeline schedule



Pipeline Schedule

VHDL Code

Figure 20: VHDL Code generation Illustration